Bujar Sefa
CSC 21200 -BC
Professor Liu
HW #1

Overview of class:
The point class is a new class that allows users to create, store, and manipulate a point in 3-dimensional space; point has an x, y, and z coordinate. The Point class can allow users to get components of points, change components of points, and even perform certain mathematical operations such as distance, checking if points make a square, etc.

7. Print():
What: This function prints out the x, y, and z coordinate of the Point that calls it.
How: This function works by using the cout function to output the Point, in which the Point x, y, z variables are accessed.
Pre-Condition: Object calling the function must be a Point.
Post-Condition: The coordinates of the Point are printed.
Worst Time Complexity: O(1)

```
void Point::print(){
  cout << "Xcoord: " << x << " YCoord: " << y
  << " Zcoord: " << z << endl;
  //NOTE not sure if it wants us to: cout << "(" << x << ", "<< y << ", " << z << ")\n";
}
```

8. Distance()
What: This function calculates the distance of a point from the origin.
How: This function works by squaring each x, y, and z coordinate. Then it adds the coordinates together, followed by the square root to provide a distance. (NOTE -- Mathematical formula of distance)
Pre-Condition: A point can only call this function (must have 3 double coordinates).
Post-Condition: The distance of the point from the origin is provided in a decimal value.
Worst Time Complexity: O(1)

```
double Point::distance(){
        double innerRoot = (x*x)+(y*y)+(z*z);
        double rad = sqrt(innerRoot)
        return rad;
}
```

9. Line()

What: This function determines if three points, one being the origin, are collinear.

How: This function works by finding a scalar of the x, y, z component between the point that calls the function and secondPoint. The scalars of the x, y, z components are then checked to see if they are equal in value. If they are equal in value, true is returned, else false is returned.

Pre-Condition: SecondPoint must be a Point with x, y, and z coordinates.

Post-Condition: Returns whether or not the two points and the origin are collinear.

Worst Time Complexity: O(1)

```
bool Point::line(Point secondPoint){
    if((x==0 && y==0 && z==0))
        return false;
    double tOfX = 0;
    double tOfY = 0;
    double tOfZ = 0;
    if(secondPoint.x != 0)
        tOfX = x/secondPoint.x;
    if(secondPoint.y !=0)
        tOfY = y/secondPoint.y;
    if(secondPoint.z != 0)
        tOfZ = z/secondPoint.z;
    double tOfX = 0;
    double tOfY = 0;
    double tOfZ = 0;
    if(secondPoint.x != 0)
        tOfX = x/secondPoint.x;
    if(secondPoint.y !=0)
        tOfY = y/secondPoint.y;
    if(secondPoint.z != 0)
        tOfZ = z/secondPoint.z;
    if(tOfX == 0 && tOfY == tOfZ){
        return true;
    }
    else if(tOfY == 0 && tOfX == tOfZ){
        return true;
    }
    else if(tOfZ == 0 && tOfX == tOfY){
        return true;
    }
    else if(tOfX == tOfY && tOfX == tOfZ){
        return true;
    }
 return false;
}
```

10. Cross()

What: This function calculates the cross product between two points.

How: This function works by using the x, y, and z coordinates of the two points and applying the cross product formula (sample <yz2-y2z,....>). It returns a new point containing the cross product (Technically a cross product is in the form of a vector).

Pre-Condition: secondPoint must be a point.

Post-Condition: The cross product of two points is returned. (A point containing the values of their cross product).

Worst Time Complexity: O(1)

```
Point Point::cross(Point secondPoint){
    double xCross = (y*secondPoint.z) - (secondPoint.y*z);
    double yCross = (z*secondPoint.x) - (x*secondPoint.z);
    double zCross = (x*secondPoint.y) - (secondPoint.x*y);
    Point crossProduct(xCross, yCross, zCross);
    return crossProduct;
}
```

11. Addition()

What: This function produces a new point that is the addition of the components of the two points(pt1 and pt2) being added.

How: This function works by taking the separate components of each of the two points(pt1,pt2)...ie(pt1.x+pt2.x), adding them together, and storing them in a new point that is returned.

Pre-Condition: Two points(pt1 and pt2) must be passed to be added.

Post-Condition: A new point is returned containing the individually added components of the two points(pt1 and pt2).

Worst Time Complexity: O(1)

```
Point operator +(const Point& pt1, const Point& pt2){
    Point added(pt1.getX()+pt2.getX(), pt1.getY()+pt2.getY(), pt1.getZ()+pt2.getZ());
    return added;
}
```

11. Subtraction()

What: This function produces a new point that is the subtraction of the components of the two points(pt1 and pt2) being subtracted.

How: This function works by taking the separate components of each of the two points(pt1,pt2)...ie(pt1.x-pt2.x), subtracting them each, and storing them in a new point that is returned.

Pre-Condition: Two points(pt1 and pt2) must be passed to be subtracted.

Post-Condition: A new point is returned containing the individually subtracted components of the two points(pt1 and pt2).
Worst Time Complexity: O(1)

```
Point operator -(const Point& pt1, const Point& pt2){
    Point added(pt1.getX()-pt2.getX(), pt1.getY()-pt2.getY(), pt1.getZ()-pt2.getZ());
    return added;
}
```

12. Input()
What: This function rewrites the input operator to take in user input of a point (all three components).
How: This function works by overloading the >> operand by letting users input points, directing and storing the values into a point object (which values are then referenced to).
Pre-Condition: Operand (>>) must be used with input to point object.
Post-Condition: The x, y, and z coordinates of point p have been read into ins.
Worst Time Complexity: O(1)

```
istream& operator >>(istream& ins, Point& p){
    ins >> p.x >> p.y >> p.z;
    return ins;
}
```

12. OutPut()
What: This function rewrites the output operator to output a point.
How: This function works by overloading the << operand by letting the point p be outputted onto the output stream outs. Individual components of point p are output one after the other.
Pre-Condition: Operand (<<) must be used with output of point object.
Post-Condition: The x, y, and z coordinates of point p have been written into outs.
Worst Time Complexity: O(1)

```
ostream& operator <<(ostream& outs, const Point& p){
    outs << p.getX() << ", " << p.getY() << ", " << p.getZ() << "\n";
    return outs;
}
```

13. Plane()
What: This function determines if a point is coplanar to 3 other points.
Why: This function works by finding the directional points between the three points in the array. Then the normal vector (in the form of a point) is calculated between the two directional points.

Using the normal vector and the fourth point, the equation of a plane is used to see if we have such an equation = 0. This comparison is then returned.
Pre-Condition: An array of 3 points, and an addition point must be passed. (The size of the array is not given, so there is no way to detect if 3 points are given. C++ limibility - Timmy).
Post-Condition: Returns if the point not in the array is on the plane formed by the 3 points in the array.
Worst Time Complexity: O(1)

```
bool plane(Point* points, Point otherPoint){
    Point u(points[1].getX() - points[0].getX(),points[1].getY() -
points[0].getY(),points[1].getZ() - points[0].getZ());
    Point v(points[2].getX() - points[0].getX(),points[2].getY() -
points[0].getY(),points[2].getZ() - points[0].getZ());
    Point normalVect = u.cross(v);
    double onPlane = (normalVect.getX()*(otherPoint.getX()-points[0].getX()))
        + (normalVect.getY()*(otherPoint.getY()-points[0].getY()))
        + (normalVect.getZ()*(otherPoint.getZ()-points[0].getZ()));
    return (onPlane == 0);

}
```

14. Square()
What: This function determines if any four points of all points given form a square.
Why: This function works by loopings through all of the points, and for every four points it checks to see if they are coplanar (using plane function). Then it checks if they have equal sides and diagonals (by comparing the distances between all the points). If all three of these cases are true, then the four points are returned. (NOTE it finds only the first case of four points, as the array may hold multiple).
Pre-Condition: Array of points must all be unique. Size must be >=4.
Post-Condition: Returns if any four points make a square or not. (NOTE if size is [0, 4), then false is returned as less than 4 points cannot make a square.)
Worst Time Complexity: O(n^4)

```
bool square(Point* points, int size){
    if(size < 4){
        return false;
    }
    for(int i = 0; i < size-3; i++){
        for(int j = i+1; j < size-2; j++){
               for(int k = j+1; k<size-1; k++){
                      for(int l = k+1; l < size; l++){
                             int threeOfFourPointsSize = 3;
                             Point threeOfFourPoints[threeOfFourPointsSize];
                             threeOfFourPoints[0] = points[j];
                             threeOfFourPoints[1] = points[k];
                             threeOfFourPoints[2] = points[l];
                             if(plane(threeOfFourPoints, points[i])){
                                    double disIToJ = points[i].distance(points[j]);
                                    double disIToK = points[i].distance(points[k]);
                                    double disIToL = points[i].distance(points[l]);
                                    double disJToK = points[j].distance(points[k]);
                                    double disJToL = points[j].distance(points[l]);
                                    double disKToL = points[k].distance(points[l]);
                                    if((disIToJ == disJToK) && (disIToJ == disIToL)){
                                           if(disIToJ == disKToL){
                                                  if(disIToK == disJToL){
                                                         return true;
                                                  }
                                           }
                                    }
                                    else if((disIToK == disJToK) && (disIToK == disIToL)){
                                           if(disIToK == disJToL){
                                                  if(disIToJ == disKToL){
                                                         return true;
                                                  }
                                           }
                                    }
                                    else if((disIToJ == disIToK) && (disIToJ == disJToL)){
                                           if(disIToJ == disKToL){
                                                  if(disIToL == disJToK){
                                                         return true;
                                                  }
                                           }
                                    }
                             }
                      }
               }
        }
    }
    return false;
}
```

15. Centroid()

What: This function calculates the centroid of an array of points.

Why: This function works by going through each point, summing up their coordinates and getting the average of each coordinate. Then it places each coordinate into a point to return.

Pre-Condition: Array must be of points. Size of array must be greater than 0.

Post-Condition: If size is <= 0, origin is returned (Technically a negative size has no centroid, but I used int for this assignment as size_t was not introduced until after). The centroid point of the points is returned.

Worst Time Complexity: O(n)

```
Point centroid(Point* points, int size){
    double xSum = 0;
    double ySum = 0;
    double zSum = 0;
    if(size<=0){
        Point p(0,0,0);
        return p;
    }
    for(int i = 0; i < size; i++){
        xSum+= points[i].getX();
        ySum+= points[i].getY();
        zSum+= points[i].getZ();
    }
    double avgX = xSum/size;
    double avgY = ySum/size;
    double avgZ = zSum/size;
    Point cent(avgX, avgY, avgZ);
    return cent;
}
```

Additional Questions/Notes:

1. Function of distance was rewritten/overridden for square():

What: Calculates the distance between two points.

How: First the function takes the difference between the x, y and z coordinates (x2-x1...). Then the differences are squared. After, each of the squared differences are added together and the square root of that whole value is taken.

Pre-Condition: secondPoint must be a point.

Post-Condition: The distance between two points is returned.

Worst Time Complexity: O(1)

```
double Point::distance(Point secondPoint){
    double innerRoot = ((secondPoint.x - x)*(secondPoint.x-x)) + ((secondPoint.y -
y)*(secondPoint.y-y)) + ((secondPoint.z - z)*(secondPoint.z-z));
    double rad = sqrt(innerRoot);
    return rad;
}
```

2. Time Complexity for square():

I'm kind of confused on the time complexity for the square function. I currently chose to stick with the solution of n^4 as there are four, four loops. I am not sure if calling the plane() function distance() within the loop would push the complexity to O(n^5). plane() and distance() are both O(1) so I would assume n*1 is just n. But being that these functions can be called many times, it might cause the complexity to increase by n. This is just some lingering thought that I would like to know more about.

3. int instead of size_t

Some functions, such as square() utilize int size instead of size_t. I feel as if using size_t would improve such functions, as it would provide more security/reassurance. With that said I stuck with int because size_t was not introduced until the very last lesson, which would apply to the following homework assignment.

Header File: Sefa_Bujar_HW1.h:

```
#ifndef __POINT_H__
#define __POINT_H__

#include <iostream>
#include <cmath> //For math functions
#include <cassert> //For assert function
using namespace std;

class Point{
  private:
       double x;
       double y;
       double z;
  public:
       double getX() const;
       double getY() const;
       double getZ() const;
       void setX(double inX); /*NOTE set function parameters might be able to be constant...*/
       void setY(double inY);
       void setZ(double inZ);
       void setXY(double inX, double inY);
       void setXYZ(double inX, double inY, double inZ);
       Point(); //Default constructor
       Point(double inX, double inY, double inZ);
       Point(double inX, double inY);
       void origin();
       Point(const Point& P); //Copy constructor
       Point& operator=(Point P); //assignment function
       void print(); //Print function
       double distance(); //Distance function
    double distance(Point secondPoint); //Overwritten distance function
    bool line(Point secondPoint); //Function that checks if points are collinear
    Point cross(Point secondPoint); //Function that gets cross product of points
    friend istream& operator >>(istream& ins, Point& p); //NOTE this is a friend function
because when inputting we need to access x,y,z.
};

Point operator +(const Point& pt1, const Point& pt2); //Overloading operator for addition.
Point operator -(const Point& pt1, const Point& pt2); //Overloading operator for subtraction.
ostream& operator <<(ostream& outs, const Point& p); //Overloading operator for output.
bool plane(Point* points, Point otherPoint); //Function that checks if a point is on a plane
bool square(Point* points, int size); //Function that checks if points make a square
Point centroid(Point* points, int size); //Function that gets the centroid of points.

#endif
```